ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY

Kanyakumari Main Road, near Anjugramam, Palkulam, Anjugramam, Tamil Nadu 629401

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

&

SMEC TECHNOLOGIES.

VALUE ADDED COURSE

ON

ANDROID APPLICATION DEVELOPEMENT

COURSE MATERIAL

INTRODUCTION

This course focuses on **Android Development**. But what is Android?

Android is an operating system. That is, it's software that connects hardware to software and provides general services. But more than that, it's a *mobile specific* operating system: an OS designed to work on *mobile* (read: handheld, wearable, carry-able) devices.

- Note that the term "Android" also is used to refer to the "platform" (e.g., devices that use the OS) as well as the ecosystem that surrounds it. This includes the device manufacturers who use the platform, and the applications that can be built and run on this platform. So "Android Development" technically means developing applications that run on the specific OS, it also gets generalized to refer to developing any kind of software that interacts with the platform.

1.1 Android History

If you're going to develop systems for Android, it's good to have some familiarity with the platform and its history, if only to give you perspective on how and why the framework is designed the way it is.

- **2003**: The platform was originally founded by a start-up "Android Inc." which aimed to build a mobile OS operating system (similar to what Nokia's Symbian was doing at the time)
- **2005**: Android was acquired by Google, who was looking to get into mobile
- **2007**: Google announces the Open Handset Alliance, a group of tech companies working together to develop "open standards" for mobile platforms. Members included phone manufacturers like HTC, Samsung, and Sony; mobile carriers like T-Mobile, Sprint, and NTT DoCoMo; hardware manufacturers like Broadcom and Nvidia; and others. The Open Handset Alliance now (2017) includes 86 companies.
  - Note this is the same year the first iPhone came out!
- **2008**: First Android device is released: the HTC Dream (a.k.a. T-Mobile G1)

  Specs: 528Mhz ARM chip; 256MB memory; 320x480 resolution capacitive touch; slide-out keyboard! Author's opinion: a fun little device.

- **2010**: First Nexus device is released: the Nexus One. These are Google-developed “flagship” devices, intended to show off the capabilities of the platform.

  Specs: 1Ghz Scorpion; 512MB memory; .37" at 480x800 AMOLED capacitive touch.

  - For comparison, the iPhone 7 Plus (2016) has: 2.34Ghz dual core A10 64bit Fusion; 3GB RAM; 5.5" at 1920x1080 display.

  As of 2016, this program has been superceded by the Pixel range of devices.

- **2014**: Android Wear, a version of Android for wearable devices (watches) is announced.
- **2016**: Daydream, a virtual reality (VR) platform for Android is announced

In short, Google keeps pushing the platform wider so it includes more and more capabilities.

Today, Android is incredibly popular (to put it mildly). Android is incredibly popular! (see e.g., here, here, and here)

- In any of these analyses there are some questions about what exactly is counted… but what we care about is that there are *a lot* of Android devices out there! And more than that: there are a lot of **different** devices!

1.1.1 Android Versions

Android has gone through a large number of “versions” since it’s release:

Date	Version	Nickname	API Level
Sep 2008	1.0	Android	1
Apr 2009	1.5	Cupcake	3
Sep 2009	1.6	Donut	4

Date	Version	Nickname	API Level
Oct 2009	2.0	Eclair	5
May 2010	2.2	Froyo	8
Dec 2010	2.3	Gingerbread	9
Feb 2011	3.0	Honeycomb	11
Oct 2011	4.0	Ice Cream Sandwich	14
July 2012	4.1	Jelly Bean	16
Oct 2013	4.4	KitKat	19
Nov 2014	5.0	Lollipop	21
Oct 2015	6.0	Marshmallow	23
Aug 2016	7.0	Nougat	24
Mar 2017	O preview	*Android O Developer Preview*	

Each different "version" is nicknamed after a dessert, in alphabetica order. But as developers, what we care about is the **API Level**, which indicates what different programming *interfaces* (classes and methods) are available to use.

- You can check out an interactive version of the history through Marshmallow at https://www.android.com/history/
- For current usage breakdown, see https://developer.android.com/about/dashboards/

Additionally, Android is an "open source" project released through the "Android Open Source Project", or ASOP. You can find the latest version of the operating system code

at https://source.android.com/; it is very worthwhile to actually dig around in the source code sometimes!

While new versions are released fairly often, this doesn't mean that all or even many devices update to the latest version. Instead, users get updated phones historically by purchasing new devices (every 18m on average in US). Beyond that, updates—including security updates—have to come through the mobile carriers, meaning that most devices are never updated beyond the version that they are purchases with.

- This is a problem from a consumer perspective, particularly in terms of security! There are some efforts on Google's part to to work around this limitation by moving more and more platform services out of the base operating system into a separate "App" called Google Play Services.
- But what this means for developers is that you can't expect devices to be running the latest version of the operating system—the range of versions you need to support is much greater than even web development!

1.1.2 Legal Battles

When discussing Android history, we would be remiss if we didn't mention some of the legal battles surrounding Android. The biggest of these is **Oracle v Google**. In a nutshell, Oracle claims that the *Java API* is copyrighted (that the method signatures themeselves and how they work are protected), so because Google uses that API in Android, Google is violating the copyright. In 2012 a California federal judge decided in Google favor (that one can't copyright an API). This was then reversed by the Federal Circuit court in 2014. The verdict was appealed to the Supreme courset in 2015, who refused to hear the case. It then went back to the the district court, which ruled that Google's use of the API was fair use. See https://www.eff.org/cases/oracle-v-google for a summary, as well as https://arstechnica.com/series/series-oracle-v-google/

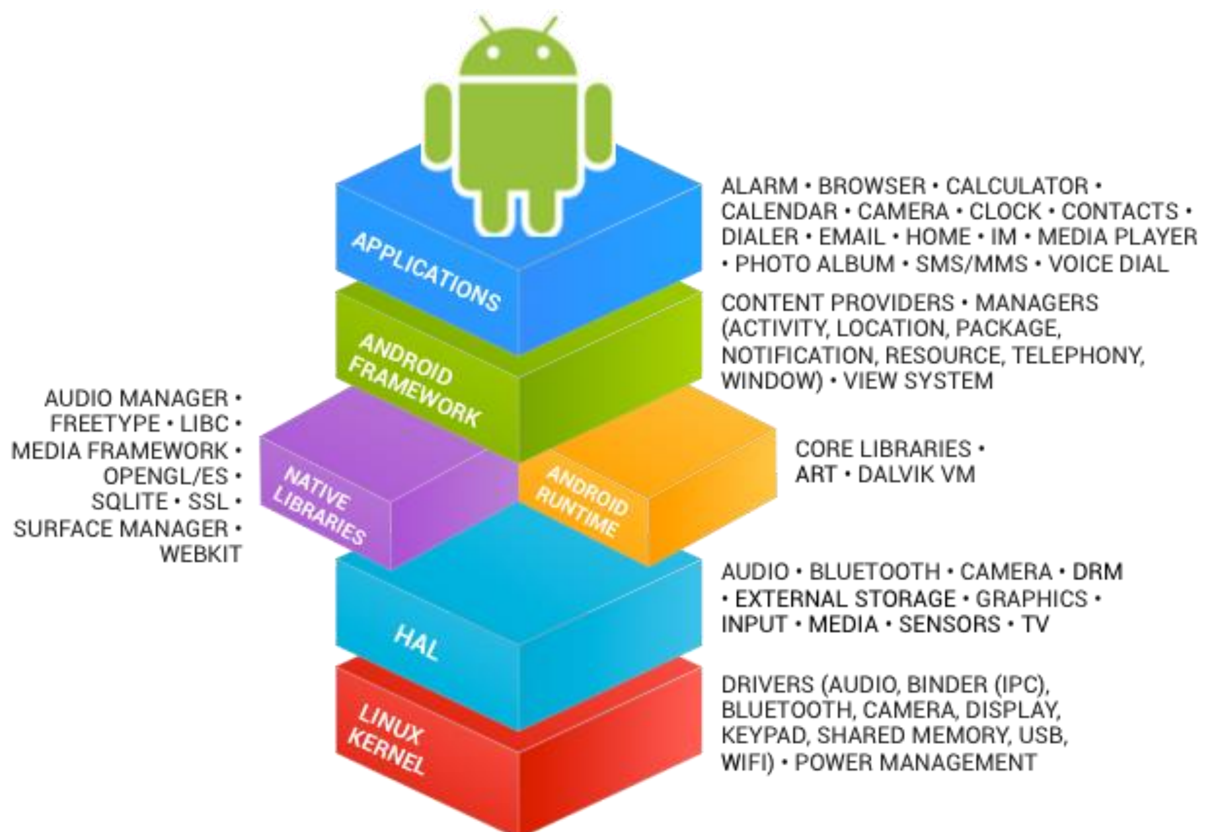
- One interesting side effect of this battle: the latest version of Android (Nougat) uses the OpenJDK implementation of Java, instead of Google's own in-violation-but-fair-use implementation see here. This change *shouldn't* have any impact on us as developers, but it's worth keeping an eye out for potentially differences between Android and Java SE.

There have been other legal challenges as well. While not directly about Android, the other major relevant court battle is **Apple v Samsung**. In this case, Apple claims that Samsung infringed on their intellectual property (their design patents). This has gone back and forth in terms of damages and what is considered infringing; the latest development is that the Supreme Court heard the case and sided with Samsung that infringing design patents shouldn't lead to damages in terms of the entire device… it's complicated (the author is not a lawyer).

So overall: Android is a growing, evolving platform that is embedded in and affecting the social infrastructures around information technology in numerous ways.

1.2 Android Architecture and Code

Developing Android applications involves interfacing with the Android platform and framework. Thus you need a high level understanding of the architecture of the Android platform. See https://source.android.com/devices/ for more details.



Android Architecture (image from: hub4tech)

Like so many other systems, the Android platform is built as a layered architecture:

- At it's base, Android runs on a Linux kernel for interacting with the device's processor, memory, etc. Thus an Android device can be seen as a Linux computer.
- On top of that kernel is the Hardware Abstraction Layer: an interface to drivers that can programmatically access hardware elements, such as the camera, disk storage, Wifi antenna, etc.
    - These drivers are generally written in C; we won't interact with them directly in this course.
- On top of the HAL is the Runtime and Android Framework, which provides a set of abstraction in the Java language which we all know an love. For this course, Android Development will involve writing Java applications that interact with the Android Framework layer, which handles the task of interacting with the device hardware for us.

1.2.1 Programming Languages

There are two programming languages we will be working with in this course:

1. **Java:** Android code (program control and logic, as well as data storage and manipulation) is written in Java.

    Writing Android code will feel a lot writing any other Java program: you create classes, define methods, instantiate objects, and call methods on those objects. But because you're working within a **framework**, there is a set of code that *already exists* to call specific methods. As a developer, your task will be to fill in what these methods do in order to run your specific application.

    - In web terms, this is closer to working with Angular (a framework) than jQuery (a library).
    - Importantly: this course expects you to have "journeyman"-level skills in Java (apprenticeship done, not yet master). We'll be using a number of intermediate concepts (like generics and inheritance) without much fanfare or explanation (though see the appendix).

2. **XML:** Android user interfaces and resources are specified in XML (E**X**tensible **M**arkup **L**anguage). To compare to web programming: the XML contains what would normally go in the HTML/CSS, while the Java code will contain what would normally go in the JavaScript.

   XML is just like HTML, but you get to make up your own tags. Except we'll be using the ones that Android made up; so it's like defining web pages, except with a new set of elements. This course expects you to have some familiarity with HTML or XML, but if not you should be able to infer what you need from the examples.

1.2.2 Building Apps

As stated above, we will write code in Java and XML. But how does that code get run on the phone's hardware?

Pre-Lollipop (5.0), Android code ran on Dalvik: a virtual machine similar to the JVM used by Java SE.

- Fun fact for people with a Computer Science background: Dalvik uses a register-based architecture rather than a stack-based one!

A developer would write *Java code*, which would then be compiled into *JVM bytecode*, which would then be translated into *DVM* (Dalvik virtual machine) bytecode, that could be run on Android devices. This DVM bytecode was stored in .dex or .odex ("[Optimized] Dalvik Executable") files, which is what was loaded onto the device. The process of converting from Jave code to dex files is called **"dexing"** (so code that has been built is "dexed").
Dalvik does include JIT ("Just In Time") compilation to native code that runs much faster than the code interpreted by the virtual machine, similar to the Java HotSpot. This navite code is faster because no translation step is needed to talk to the actual hardware (the OS).

From Lollipop (5.0) on, Android instead uses Android Runtime (ART) to run code. ART's biggest benefit is that it compiles the .dex bytecode into native code *on installation* using AOT ("Ahead of Time") compilation. ART continues to accept .dex bytecode for backwards compatibility (so the same dexing process occurs), but the code that is actually installed and run on a device is native. This allows for applications to have faster execution, but at the

cost of longer install times—but since you only install an application once, this is a pretty good trade.

After being built, Android applications (the source, dexed bytecode, and any resources) are packaged into **.apk** files. These are basically zip files (they use the same gzip compression); if you rename the file to be .zip and you can unpackage them! The .apk files are then cryptographically signed to specify their authenticity, and either "side-loaded" onto the device or uploaded to an App Store for deployment.

- The signed .apk files are basically the "executable" versions of your program!
- Note that the Android application framework code is actually "pre-DEXed" (pre-compiled) on the device; when you write code, you're actually compiling against empty code stubs (rather than needing to include those classes in your .apk)! That said, any other 3rd-party libraries you include will be copied into your built App, which can increase its file size both for installation and on the device.

To summarize, in addition to writing Java and XML code, when building an App you need to:

1. Generate Java source files (e.g., from resource files, which are written XML used to generate Java code)
2. Compile Java code into JVM bytecode
3. "dex" the JVM bytecode into Dalvik bytecode
4. Pack in assets and graphics into an APK
5. Cryptographically sign the APK file to verify it
6. Load it onto the device

There are a lot of steps here, but there are tools that take care of it for us. We'll just write Java and XML code and run a "build" script to do all of the steps!

1.3 Development Tools

There are a number of different hardware and software tools you will need to do Android development:

1.3.1 Hardware

Since Android code is written for a virtual machine anyway, Android apps can be developed and built on any computer's operating system (unlike some other mobile OS…).

But obviously Android apps will need to be run on Android devices. Physical devices are the best for development (they are the fastest, easiest way to test), though you'll need USB cable to be able to wire your device into your computer. Any device will work for this course; you don't even need cellular service (just WiFi should work). Note that if you are unfamiliar with Android devices, you should be sure to play around with the interface to get used to the interaction language, e.g., how to click/swipe/drag/long-click elements to use an app.

- You will need to turn on developer options in order to install development apps on your device!

If you don't have a physical device, it is also possible to use the Android Emulator, which is a "virtual" Android device. The emulator represents a generic device with hardware you can specify… but it does have some limitations (e.g., no cellular service, no bluetooth, etc).

- While it has improved recently, the Emulator historically does not work very well on Windows; I recommend you develop on either a Mac or a physical device. In either case, make sure you have enabled HAXM (Intel's Acceleration Manager, which allows the emulator to utilize your GPU for rendering): this speeds things up considerably.

1.3.2 Software

Software needed to develop Android applications includes:

- The Java 7 **SDK** (not just the JRE!) This is because you're writing Java code!
- Gradle or Apache ANT. These are *automated build tools*—in effect, they let you specify a single command that will do a bunch of steps at once (e.g., compile files, dex files, move files, etc). These are how we make the "build script" that does the 6 build steps listed above.

    - ANT is the "old" build system, Gradle is the "modern" build system (and so what we will be focusing on).
    - Note that you do not need to install Gradle separately for this course.

- Android Studio & Android SDK is the official IDE for developing Android applications. Note that the IDE comes bundled with the SDK. Android Studio provides the main build system: all of the other software (Java, Gradle) goes to support this.

  The SDK comes with a number of useful command-line tools. These include:

    - `adb`, the "Android Device Bridge", which is a connection between your computer and the device (physical *or* virtual). This tool is used for console output!
    - `emulator`, which is a tool used to run the Android emulator
    - *deprecated/removed* `android`: a tool that does SDK/AVD (Android Virtual Device) management. Basically, this command-line utility did everything that the IDE did, but from the command-line! It has recently been removed from the IDE.

  I recommend making sure that the SDK command-line tools are installed. Put the `tools` and `platform-tools` folders on your computer's `PATH`; you can run `adb` to check that everything works. All of these tools are built into the IDE, but they can be useful fallbacks for debugging.

1.4 Hello World

As a final introductory steps, this lecture will walk you through creating and running a basic App so that you can see what you will actually be working with. You will need to have Android Studio installed for this to work.

1. Launch Android Studio if you have it (may take a few minutes to open)
2. Start a new project.
    - Use your UW NetID in the domain.
    - Make a mental note of the project location so you can find your code later!

    - *Target*: this is the "minimum" SDK you support. We're going to target Ice Cream Sandwich (4.0.3, API 15) for most this class, as the earliest version of Android most our apps will support.

        - Note that this is different than the "target SDK", which is the version of Android you tested your application against (e.g., what system did you run it on?) For this course we will be testing on API 21 (Lollipop); we'll specify that in a moment.

3. Select an *Empty Activity*

    - **Activities** are "Screens" in your application (things the user can do). Activities are discussed in more detail in the next lecture.

4. And boom, you have an Android app! Aren't frameworks lovely?

1.4.1 The Emulator

We can run our app by clicking the "Play" or "Run" button at the top of the IDE. But we'll need a device to run the app on, so let's make an emulator!

The **Nexus 5** is a good choice for supporting "older" devices. The new Pixel is also a reasonable device to test against.

- You'll want to make sure you create a Lollipop device, using the Google APIs (so we have special classes available to us), and amost certainly running on x86 (Intel) hardware

- Make sure that you've specified that it accepts keyboard input. You can always edit this emulator later (Tools > Android > AVD Manager).

After the emulator boots, you can slide to unlock, and there is our app!

1.4.2 Project Contents

So what does our app look like in code? What do we have?

Note that Android Studio by default shows the **"Android"** view, which organizes files thematically. If you go to the **"Project"** view you can see what the actual file system looks like. In Android view, files are organized as follows:

- app/ folder contains our application
    - manifests/ contains the **Android Manifest** files, which is sort of like a "config" file for the app
    - java/ contains the Java source code for your project. You can find the MyActivity file in here
    - res/ contains resource files used in the app. These are where we're going to put layout/appearance information
- Also have the Gradle scripts. There are a lot of these:
    - build.gradle: Top-level Gradle build; project-level (for building!)
    - app/build.gradle: Gradle build specific to the app **use this one to customize project!**. We can change the *Target SDK* in here!
    - proguard-rules.pro: config for release version (minimization, obfuscation, etc).
    - gradle.properties: Gradle-specific build settings, shared
    - local.properties: settings local to this machine only
    - settings.gradle: Gradle-specific build settings, shared

  Note that ANT would instead give:

    - build.xml: Ant build script integrated with Android SDK
    - build.properties: settings used for build across all machines
    - local.properties: settings local to this machine only

  We're using Gradle, but it is good to be aware of ANT stuff for legacy purposes

- res has resource files. These are **XML** files that specify details of the app–such as layout.
    - res/drawable/: contains graphics (PNG, JPEG, etc)
    - res/layout/: contains UI XML layout files
    - res/mipmap/: conatins launcher icon files in different resolutions
        - Fun fact: MIP stands for "*multum in parvo*", which is Latin for "much in little" (because multiple resolutions of the images are stored in a single file). "Map" is used because Mipmaps are normally used for texture mapping.
    - res/values/: contains XML definitions for general constants

See also: http://developer.android.com/guide/topics/resources/available-resources.html, or Lecture 3.

We can also consider what the application code does. While we'll revisit this in more detail in the next lecture, it's useful to start seeing how the framework is structured:

We'll start with the **MyActivity** Java source file. This class extends Activity (actually it extends a subclass that supports Material Design components), allowing us making our own customizations to what the app does.
In this class, we override the onCreate() method that is called by the framework when the Activity starts (see next lecture).

- We call the super method, and then setContentView() to specify what the content (appearance) of our Activity is. This is passed in a value from something called R. R is a class that is **generated at compile time** and contains constants that are defined by the XML "resource" files! Those files are converted into Java variables, which we can access through the R class.

R.layout refers to the "layout" XML resource, so can go there (remember: inside res/). Opening these XML files they appear in a "design" view. This view lets you use a graphical system to lay out your application (similar to a PowerPoint slide).

- However, even as the design view becomes more powerful, using it is still frowned upon by many developers for historical reasons. It's often cleaner to write out the layouts and content in code. This is the same difference between writing your own HTML and using something like FrontPage or DreamWeaver or Wix to create a page. While those are legitimate applications, they are less "professional".

In the code view, we can see the XML: tags, attributes, values. Tags nested inside one another. The provided XML code defines a layout, and inside that is a TextView (a View representing some text), which has a value: text! We can change that and then *re-run the app* to see it update!

- It's also possible to define this value in values/strings (e.g., as a constant), then refer to as @string/message. More on this proces later.

Finally, as a fun demonstration, try to set an icon for the App (in Android Studio, go to: File > New > Image Asset)

2.1 Resources

Resources can be found in the **res/** folder, and represent elements or data that are "external" to the code. You can think of them as "media content": often images, but also things like text clippings (or short String constants). Textual resources are usually defined in XML files. This is because resources represent elements (e.g., content) that is *separate* from the code (the behavior of the app), so is kept separate from the Java code to support the **Principle of Separation of Concerns**

- By defining resources in XML, they can be developed (worked on) *without* coding tools (e.g., with systems like the graphical "layout design" tab). Theoretically you could have a Graphic Designer create these resources, which can then be integrated into the code without the designer needing to do a lick of Java.

- Similarly, keeping resources separate allows you to choose what resources to include *dynamically*. You can choose to show different images based on device screen resolution, or pick different Strings based on the language of the device (internationalization!)—the behavior of the app is the same, but the "content" is different!

    - This is similar to how in web development we may want to have the same JavaScript from different HTML.

What should be a resource? In general:

- Layouts should **always** be resources

- UI controls (buttons, etc) should *mostly* be defined as resources (part of layouts), though behavior will be defined programmtically (in Java)

- Any graphic images (drawables) should be resources

- Any *user-facing* strings should be resources

- Style and theming information should be resources

As introduced in Lecture 1, there are a number of different resource types used in Android, many of which can be found in the res/ folder of a default Android project, including:

- res/drawable/: contains graphics (PNG, JPEG, etc)

- res/layout/: contains UI XML layout files
- res/mipmap/: conatins launcher icon files in different resolutions
- res/values/: contains XML definitions for general constants
    - /strings: short string constants (e.g., labels)
    - /colors: color constants
    - /styles : constants for style and theme details
    - /dimen : dimensional constants (like default margins); not created by default in Android Studio 2.3+.

The details about these different kinds of resources is a bit scattered throughout the documentation, but Resource Types[6] is a good place to start, as is Providing Resources.

2.1.1 Alternate Resources

These aren't the only names for resource folders: as mentioned above, part of the goal of resources is that they can be **localized**: changed depending on the device! You are thus able to specify folders for "alternative" resources (e.g., special handling for another language, or for low-resolution devices). At runtime, Android will check the configuration of the device, and try to find an alternative resource that matches that config. If it it *can't* find a relevant alternative resource, it will fall back to the "default" resource.

There are many different configurations that can be used to influence resources; see Providing Resources[7]. To highlight a few options, you can specify different resources based on:

- Language and region (e.g., via two-letter ISO codes)

- Screen size(small, normal, medium, large, xlarge)
- Screen orientation (port for portrait, land for landscape)
- Specifc screen pixel density (dpi) (ldpi, mdpi, hdpi, xhdpi, xxhdpi, etc.). xxhdpi is pretty common for high-end devices. Note that dpi is "dots per inch", so these values represent the number of pixels across *relative* to the device size!
- Platform version (v1, v4, v7… for each API number)

Configurations are indicated using the **directory name**, giving them the form <resource_name>(-<config_qualifier>)+

- You can see this in action by using the *New Resource* wizard (File > New > Android resource file) to create a welcome message (a string resource, such as for

the app_name) in another language[8], and then changing the device’s language settings to see the content automatically adjust!

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">Mon Application</string>
</resources>
```

- Switch to the Package view in Android Studio to see how the folder structure for this works.

2.1.2 XML Details

Resources are usually defined as XML (which is similar in syntax to HTML). The strings.xml example used above involves fairly simple elements but more complexresource is pretty simple, but more complex details can be seen in the activity_main.xml resource inside layout/.

- Android-specific attributes are namespaced with a android: prefix, to avoid any potential conflicts (e.g., so we know we’re talking about Android’s text instead of something else).
- We can use the @ symbol to reference one resource from another, following the schema @[<package_name>:]<resource_type>/<resource_name>
- We can also use the + symbol to create a *new* resource that we can refer to; this is a bit like declaring a variable inside an attribute. This is most commonly used with the android:id attribute (android:id="@+id/identifier"), see below for details.

2.1.3 R

Although XML resources are defined separately from the Java code, resources can be accessed from within Java. When an application is compiled, the build tools (e.g., gradle) **generate** an additional Java class called **R** (for “resource”). This class contains what is basically a giant list of static “constants”—one for each resource! These constants are organized into subclasses, one for each resource type. This allows you to refer to a specific resource in the Java code as [(package_name).]R.resource_type.identifier similar to the kind of syntax used to refer to a nested JSON object! For example: R.string.hello (the hello string resource), R.drawable.icon or R.layout.activity_main

- For most resources, the identifier is defined as an element attribute (id for specific View elements in layouts, name attribute for values). For more complex resources such as entire layouts or drawables, the identifier is the *filename* (without the XML); hence R.layout.activity_main refers to the root element of the layout/activity_main.xml file.
- Note that that @ symbol used in the XML goes to the R Java file to look things up, so follows the same reference syntax.

You can find the generated R.java file inside app/build/generated/source/r/debug/... (Use the Project Files view in Android Studio).

The static constants inside the R.java file are often just ints that are *pointers* to element references (similar to passing a pointer* around in the C language). So in the Java, we usually work with int as the data type for XML resources, because we're actually working with pointers *to* those resources.

- You can think of each int constant as a "key" or "index" for that resource (in the list of all resources). Android does the hard work of taking that int, looking it up in an internal resource table, finding the associated XML file, and then getting the right element out of that XML. (By hard work, I mean in terms of implementation. Android is looking up these references directly in memory, so the look-up is a fast O(1)).

Because the R class is included in the Java, we can access these constants directly in our code (as R.resource_type.identifier). For example, the setContentView() call in an Activity's onCreate() takes in a resource int.

- The other comment method that utilizes resources will be findViewById(int), which is used to reference a View element (e.g., a button) from the resource in order to call methods on it in Java. This is the same method used with the Button example in the Activities lecture

The R class is regenerated all time (any time you change a resource, which is often); when Eclipse was the recommend Android IDE, you often needed to manually regenerate the class so that the IDE's index would stay up to date! You can perform a similar task in Android Studio by using Build > Clean Project and Build > Rebuild Project.

2.2 Views

The most common type of element we'll define in resources are **Views**[9]. `View` is the superclass for visual interface elements—a visual component on the screen is a View. Specific types of Views include: TextViews, ImageViews, Buttons, etc.

- `View` is a superclass for these components because it allows us to use **polymorphism** to treat all these visual elements the same way as instances of the same type. We can lay them out, draw them, click on them, move them, etc. And all the behavior will be the same—though subclasses can also have "extra" features

Here's the big trick: one subclass of `View` is `ViewGroup`[10]. A `ViewGroup` can contain other "child" Views. But since `ViewGroup` is a `View`… it can contain more `ViewGroups` inside it! Thus we can **nest** Views within Views, following the Composite Pattern. This ends up working a lot like HTML (which can have DOM elements like `<div>` inside other DOM elements), allowing for complex user interfaces.

- Thus Views are structured into a *tree*, what is known as the **View hierarchy**.

Views are defined inside of Layouts—that is, inside a layout resource, which is an XML file describing Views. These resources are "inflated" (rendered) into UI objects that are part of the application.

Technically, `Layouts` are simply `ViewGroups` that provide "ordering" and "positioning" information for the Views inside of them. they let the system "lay out" the Views intelligently and effectively. *Individual views shouldn't know their own position*; this follows from good good object-oriented design and keeps the Views encapsulated.

Android studio does come with a graphical Layout Editor (the "Design" tab) that can be used to create layouts. However, most developers stick with writing layouts in XML. This is mostly because early design tools were pathetic and unusable, so XML was all we had. Although Android Studio's graphical editor can be effective, for this course you should create layouts "by hand" in XML. This is helpful for making sure you understand the pieces underlying development, and is a skill you should be comfortable with anyway (similar to how we encourage people to use `git` from the command-line).

2.2.1 View Properties

Before we get into how to group Views, let's focus on the individual, basic View classes. As an example, consider the activity_main layout in the lecture code. This layout contains two individual View elements (inside a Layout): a TextView and a Button.

All View have **properties** which define the state of the View. Properties are usually defined within the resource XML as element *attributes*. Some examples of these property attributes are described below.

- **android:id** specifies a unique identifier for the View. This identifier needs to be unique within the layout, though ideally is unique within the entire app (for clarity). Identifiers must be legal Java variable names (because they are turned into a variable name in the R class), and by convention are named in lower_case format.

    - *Style tip*: it is useful to prefix each View's id with its type (e.g., btn, txt, edt). This helps with making the code self-documenting.

  You should give each interactive View a unique id, which will allow its state to automatically be saved as a Bundle when the Activity is destroyed. See here for details.
- **android:layout_width** and **android:layout_height** are used to specify the View's size on the screen (see ViewGroup.LayoutParams for documentation). These values can be a specific value (e.g., 12dp), but more commonly is one of two special values:
    - wrap_content, meaning the dimension should be as large as the content requires, plus padding.
    - match_parent, meaning the dimension should be as large as the *parent* (container) element, minus padding. This value was renamed from fill_parent (which has now been deprecated).

Android utilizes the following dimensions or units:

- **dp** is a "density-independent pixel". On a 160-dpi (dots-per-inch) screen, 1dp equals 1px (pixel). But as dpi increases, the number of pixels per dp increases. These values should be used instead of px, as it allows dimensions to work independent of the hardware's dpi (which is *highly* variable).
- **px** is an actual screen pixel. *DO NOT USE THIS* (use dp instead!)

- **sp** is a “scale-independent pixel”. This value is like dp, but is scale by the system’s font preference (e.g., if the user has selected that the device should display in a larger font, 1sp will cover more dp). *You should **always** use sp for text dimensions, in order to support user preferences and accessibility.*
- **pt** is 1/72 of an inch of the physical screen. Similar units mm and in are available. *Not recommended for use.*

- **android:padding**, **android:paddingLeft**, **android:margin**, **android:marginLeft**, etc. are used to specify the margin and padding for Views. These work basically the same way they do in CSS: padding is the space between the content and the “edge” of the View, and margin is the space between Views. Note that unlike CSS, margins between elements do not collapse.
- **android:textSize** specifies the “font size” of textual Views (use sp units!), **android:textColor** specifies the color of text (reference a color resource!), etc.
- There are lots of other properties as well! You can see a listing of generic properties in the View[11] documentation, look at the options in the “Design” tab of Android Studio, or browse the auto-complete options in the IDE. Each different View class (e.g., TextView, ImageView, etc.) will also have their own set of properties.

Note that unlike CSS, styling properties specified in the layout XML resources are not inherited; we’re effectively specifying an inline style attribute for that element, and one that won’t affect child elements. In order to define shared style properies, you’ll need to use styles resources, which are discussed in a later lecture.

While it is possible to specify these visual properties dynamically via Java methods (e.g., setText(), setPadding()). You should **only** use Java methods to specify View properties when they *need* to be dynamic (e.g., the text changes in response to a button click)—it is much cleaner and effective to specify as much visual detail in the XML resource files as possible. It’s also possible to simply replace one layout resource with another (see below).

- Views also have inspection methods such as isVisible() and hasFocus(); we will point to those as we need them.

Do not define Views or View appearances in an Activity's onCreate() callback, unless the properties (e.g., content) truly cannot be determined before runtime! Specify layouts in the XML instead.

2.2.2 Practice

Add a new ImageView element that contains a picture. Be sure and specify its id and size (experiment with different options).

You can specify the content of the image in the XML resource using the android:src attribute (use @ to reference a drawable), or you can specify the content dynamically in Java code:

```
ImageView imageView = (ImageView)findViewById(R.id.img_view);
imageView.setImageResource(R.drawable.my_image);
```

2.3 Layouts

As mentioned above, a Layout is a grouping of Views (specifically, a ViewGroup). A Layout acts as a container for other Views, to help organize things. Layouts are all subclasses of ViewGroup, so you can use its inheritance documentation to see a (mostly) complete list of options, though many of the listed classes are deprecated in favor of later, more generic/powerful options.

2.2.1 LinearLayout

Probably the simplest Layout to understand is the LinearLayout. This Layout simply orders the children View in a line ("linearly"). All children are laid out in a single direction, but you can specify whether this is horizontal or vertical with the android:orientation property. See LinearLayout.LayoutParams for a list of all attribute options!

- Remember: since a Layout is a ViewGroup is a View, you can also utilize all the properties discussed above; the attributes are inherited!

Another common property you might want to control in a LinearLayout is how much of any remaining space the elements should occupy (e.g., should they expand). This is done with the android:layout_weght property. After all element sizes are calculated (via their individual properties), the remaining space within the Layout is divided up proportionally

to the layout_weight of each element (which defaults to 0 so they get no extra space). See the example in the guide for more details.

- *Useful tip*: Give elements 0dp width or height and 1 for weight to make everything in the Layout the same size!

You can also use the android:layout_gravity property to specify the "alignment" of elements within the Layout (e.g., where they "fall" to). Note that this property is specified on individual child Views.

An important point Since Layouts *are* Views, you can of course nest LinearLayouts inside each other! So you can make "grids" by creating a vertical Layout containing "rows" of horizontal Layouts (which contain Views). As with HTML, there are lots of different options for achieving any particular interface layout.

2.2.2 RelativeLayout

A RelativeLayout is more flexible (and hence powerful), but can be more complex to use. In a RelativeLayout, children are positioned "relative" to the parent **OR** *to each other*. All children default to the top-left of the Layout, but you can give them properties from RelativeLayout.LayoutParams to specify where they should go instead.

For example: android:layout_verticalCenter centers the View vertically within the parent. android:layout_toRightOf places the View to the right of the View with the given resource id (use an @ reference to refer to the View by its id):

```
<TextView
   android:id="@+id/first"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:text="FirstString" />
<TextView
   android:id="@+id/second"
   android:layout_height="wrap_content"
   android:layout_below="@id/first"
   android:layout_alignParentLeft="true"
   android:text="SecondString" />
```

(Recall that the @+ syntax defines a *new* View id, like declaring a variable!)

You do not need to specify both toRightOf and toLeftOf; think about placing one element on the screen, then putting another element relative to what came before. This can be tricky. For this reason the author prefers to use LinearLayouts, since you can always produce a Relative positioning using enough LinearLayouts (and most layouts end up being linear in some fashion anyway!)

2.2.3 ConstraintLayout

ConstraintLayout is a Layout provided as part of an extra support library, and is what is used by Android Studio's "Design" tool (and thus is the default Layout for new layout resources). ConstraintLayout works in a manner conceptually similar to RelativeLayout, in that you specify the location of Views in relationship to one another. However, ConstraintLayout offers a more powerful set of relationships in the form of *constraints*, which can be used to create highly responsive layouts. See the class documentation for more details and examples of constraints you can add.
The main advantage of ConstraintLayout is that it supports development through Android Studio's Design tool. However, since this course is focusing on implementing the resource XML files rather than using the specific tool (that may change in a year's time), we will primarily be using other layouts.

2.2.4 Other Layouts

There are many other layouts as well, though we won't go over them all in depth. They all work in similar ways; check the individual class's documentatoion for details.

- FrameLayout is a sort of "placeholder" layout that holds a **single** child View (a second child will not be shown). You can think of this layout as a way of adding a simple container to use for padding, etc. It is also highly useful for situations where the framework requires you to specify a Layout resource instead of just an individual View.

- GridLayout arranges Views into a Grid. It is similar to LinearLayout, but places elements into a grid rather than into a line.

  Note that this is different than a Grid_View_, which is a scrollable, adaptable list (similar to a ListView, which is discussed in the next lecture).

- TableLayout acts like an HTML table: you define TableRow layouts which can be filled with content. This View is not commonly used.

2.2.5 Combining and Inflating Layouts

It is possible to combine multiple layout resources. This is useful if you want to dynamically change what Views are included, or to refactor parts of a layout into different XML files to improve code organization.

As one option, you can *statically* include XML layouts inside other layouts by using an <include> element:

```
<include layout="@layout/sub_layout">
```

But it is also possible to dynamically load views “manually” (e.g., in Java code) using the LayoutInflator. This is a class that has the job of “inflating” (rendering) Views. The process is called “inflating” based on the idea that it is “unpacking” or “expanding” a compact resource description into a complex Java Object. LayoutInflator is implicitly used in the setContentView() method, but can also be used independently with the following syntax:

```
LayoutInflator inflator = getLayoutInflator(); //access the inflator (called on the Activity)
View myLayout = inflator.inflate(R.layout.my_layout, parentViewGroup, true); //to attach
```

Note that we never instantiate the LayoutInflator, we just access an object that is defined as part of the Activity.

The inflate() method takes a couple of arguments:

- The first parameter is a reference to the resource to inflate (an int saved in R)
- The second parameter is a ViewGroup to act as the “parent” for this View—e.g., what layout should the View be inflate inside? This can be null if there is not yet a layout context; e.g., you wish to inflate the View but not show it on the screen yet.
- The third (optional) parameter is whether to actually attach the inflated View to that parent (if not, the parent just provides context and layout params to use). If not assigning to parent on inflation, you can later attach the View using methods in ViewGroup (e.g., addView(View) similar to what we’ve done with Swing).

Manually inflating a View works for dynamically loading resources, and we will often see UI implementation patterns that utilize Inflators.

However, for dynamic View creation it tends to be messy and hard to maintain (UI work should be specified entirely in the XML, without needing multiple references to parent and child Views) so it isn't as common in modern development. A much cleaner solution is to use a ViewStub[12]. A ViewStub is like an "on deck" Layout: it is written into the XML, but isn't actually shown until you choose to reveal it via Java code. With a ViewStub, Android inflates the View at runtime, but then removes it from the parent (leaving a "stub" in its place). When you call inflate() (or setVisible(View.VISIBLE)) on that stub, it is reattached to the View tree and displayed:

```
<!-- XML -->
<ViewStub android:id="@+id/stub"
    android:inflatedId="@+id/subTree"
    android:layout="@layout/mySubTree"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
//Java
ViewStub stub = (ViewStub)findViewById(R.id.stub);
View inflated = stub.inflate();
```

3.1 Inputs

The previous lecture discussed **Views** and ViewGroups (**Layouts**), and introduced some basic Views such as TextView, ImageView, and Button.

A Button is an example of an Input Control. These are simple (single-purpose; not necessarily lacking complexity) widgets that allow for user input. There are many such widgets in addition to Button, mostly found in the android.widget package. Many correspond to HTML <input> elements, but Android provided additional widgets at well.

Launch the lecture code's MainActivity with a content View of R.id.input_control_layout to see an example of many widgets (as well as a demonstration of a more complex layout!). These widgets include:

- Button, a widget that affords clicking. Buttons can display text, images or both.

- EditText, a widget for user text entry. Note that you can use the android:inputType property to specify the type of the input similar to an HTML <input>.
- Checkbox, a widget for selecting an on-off state
- RadioButton, a widget for selecting from a set of choices. Put RadioButton elements inside a RadioGroup element to make the buttons mutually exclusive.
- ToggleButton, another widget for selecting an on-off state.
- Switch, yet another widget for selecting an on-off state. This is just a ToggleButton with a slider UI. It was introduced in API 14 and is the "modern" way of supporting on-off input.
- Spinner, a widget for picking from an array of choices, similar to a drop-down menu. Note that you should define the choices as a resource (e.g., in strrings.xml).
- Pickers: a compound control around some specific input (dates, times, etc). These are typically used in pop-up dialogs, which will be discussed in a future lecture.
- …and more! See the android.widget package for further options.

All these input controls basically work the same way: you define (instantiate) them in the layout resource, then access them in Java in order to define interaction behavior.

There are two ways of interacting with controls (and Views in general) from the Java code:

1. Calling **methods** on the View to manipulate it. This represents "outside to inside" communication (with respect to the View).
2. Listening for **events** produced by the View and responding to then. This represents "inside to outside" communication (with respect to the View).

An example of the second, event-driven approach was introduced in Lecture 2. This involved *registering a listener* for the event (after acquiring a reference to the View with findViewById()) and then specifying a **callback method** (by instantiating the Listener interface) that wiould be "called back to" when the event occurs.

- It is also possible to specify the callback method in the XML resource itself by using e.g., the android:onClick attribute. This value of this attribute should be the *name* of the callback method: It is also possible to
- **<Button**

```
•     android:layout_width="wrap_content"
•     android:layout_height="wrap_content"
      android:onClick="handleButtonClick" />
```

The callback method is declared in the Java code as taking in a View parameter (which will be a reference to whatever View caused the event to occur) and returning void:

```
public void handleButtonClick(View view) { }
```

- We will utilize a mix of both of these strategies (defining callbacks in both the Java and the XML) in this class.

> *Author's Opinion*: It is arguable about which approach is "better". Specifying the callback method in the Java code helps keep the appearance and behavior separate, and avoids introducing hidden dependencies for resources (the Activity must provide the required callback). However, as buttons are made to be pressed, it isn't unreasonable to give a "name" in the XML resource as to what the button will do, especially as the corresponding Java method may just be a "launcher" method that calls something else. Specifying the callback in the XML resource may often seem faster and easier, and we will use whichever option best supports clarity of our code.

Event callbacks are used to respond to all kind of input control widgets. CheckBoxes use an onClick callback, ToggleButtons use onCheckedChanged, etc. Other common events can be found in the View documentation, and are handled via listeners such as OnDragListener (for drags), OnHoverListener (for "hover" events), OnKeyListener (for when user types), or OnLayoutChangeListener (for when layout changes display).
In addition to listening for events, it is possible to call methods directly on referenced Views to access their state. In addition to generic View methods such as isVisible() or hasFocus(), it is possible to inquire directly about the state of the input provided. For example, the isChecked() method returns whether or not a checkbox is ticked.
This is also a good way of getting access to inputted content from the Java Code. For example, call getText() on an EditText control in order to fetch the contents of that View.

- For practice, try to log out the contents of the included EditText control when the Button is pressed!

Between listening for events and querying for state, we can fully interact with input controls. Check the official documentation for more details on how to use specific individual widgets.

3.2 ListViews and Adapters

The remainder of the lecture utilizes the list_layout Layout in the lecture code. Modify MainActivity so that it uses this resource as its viewContent.

Having covered basic controls, this section will now look at some more advanced interactive Views. In particular, it will discuss how to utilize a ListView[13], which is a ViewGroup that displays a scrollable list of items! A ListView is basically a LinearLayout inside of a ScrollView (which is a ViewGroup that can be scrolled). Each element within the LinearLayout is another View (usually a Layout) representing a particular item in a list.
But the ListView does extra work beyond just nesting Views: it keeps track of what items are already displayed on the screen, inflating only the visible items (plus a few extra on the top and bottom as buffers). Then as the user scrolls, the ListView takes the disappearing views and *recycles* them (altering their content, but not reinflating from scratch) in order to reuse them for the new items that appear. This lets it save memory, provide better performance, and overall work more smoothly. See this tutorial for diagrams and further explanation of this recycling behavior.

- Note that a more advanced and flexible version of this behavior is offered by the RecyclerView. See also this guide for more details.

The ListView control uses a **Model-View-Controller (MVC)** architecture. This is a deisgn pattern common to UI systems which organizes programs into three parts:

1. The **Model**, which is the data or information in the system
2. The **View**, which is the display or representation of that data
3. The **Controller**, which acts as an intermediary between the Model and View and hooks them together.

The MVC pattern can be found all over Android. At a high level, the resources provide *models* and *views* (separately), while the Java Activities act as *controllers*.

- *Fun fact*: The Model-View-Controller pattern was originally developed as part of the Smalltalk language, which was the first Object-Oriented language!

Thus in order to utilize a ListView, we'll have some data to be displayed (the *model*), the *views* (Layouts) to be shown, and the ListView itself will connect these together act as the *controller*. Specifically, the ListView is a subclass of AdapterView, which is a View backed by a data source—the AdapterView exists to hook the View and the data together (a controller!)

- There are other AdapterViews as well. For example, GridView works exactly the same way as a ListView, but lays out items in a scrollable grid rather than a scrollable list.

In order to use a ListView, we need to get the pieces in place:

1. First we specify the **model**: some raw data. We will start with a simple String[], filling it with placeholder data:

```
2. String[] data = new String[99];
3. for(int i=99; i>0; i--){
4.     data[99-i] = i+ " bottles of beer on the wall";
   }
```

   While we could define this data as an XML resource, we'll create it dynamically for testing (and to make it changeable later!)

5. Next we specify the **view**: a View to show for each datum in the list. Define an XML layout resource for that (list_item is a good name and a common idiom).
   For simplicity's sake we don't need to specify a full Layout, just a basic TextView. Have the width match_parent and the height wrap_content. *Don't forget an id!*

```
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/txtItem"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

   To make it look better, you can specify android:minHeight="?android:attr/listPreferredItemHeight" (using the framework's preferred height for lists), and some center_vertical gravity. The android:lines property is also useful if you need more space.

6. Finally, we specify the **controller**: the ListView itself. Add that item to the Activity's Layout resource (*practice*: what should its dimensions be?)

To finish the controller ListView, we ned to provide it with an Adapter[14] which will connect the *model* to the *view*. The Adapter does the "translation" work between model and view, performing a mapping from data types (e.g., a String) and View types (e.g., a TextView). Specifically, we will use an ArrayAdapter, which is one of the simplest Adapters to use (and because we have an array of data!) An ArrayAdapter creates Views by calling .toString() on each item in the array, and setting that String as the content of a TextView!

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
    R.layout.list_item_layout, R.layout.list_item_txtView, myStringArray);
```

- Note the parameters of the constructor: a Context, the item layout resource, the TextView reource, and the data array. Also note that this instance utilizes *generics*, since we're using an array of Strings (as opposed to an array of Dogs or some other type).

We acquire a reference to the ListView with findViewById(), and call ListView#setAdapter() to attach the adapter to that controller.

```
ListView listView = (ListView)findViewById(R.id.listview);
listView.setAdapter(adapter);
```

And that's all that is needed to create a scrollable list of data!

Each item in this list is selectable (can have an onClick callback). This allows us to click on any item in order to (for example) view more details about the item. Utilize the AdapterView#setOnItemClickListener(OnItemClickListener) function to register the callback.

- The postion parameter in the onItemClick() callback is the index of the item which was clicked. Use (Type)parent.getItemAtPosition(position) to access the data value associated with that View.

Additionally, each item does have an individual layout, so we can customize these appearances (e.g., if our layout also wanted to include pictures). See this tutorial for an example on making a custom adapter to fill in multiple Views with data from a list!

And remember, a GridView is basically the same thing (in fact, we can just change over that and have everything work, if we use *polymorphism*!)

3.3 Network Data

In the previous section we created a ListView utilizing an adapter to display a list of Strings. But Appendix C provides an implementation for fetching data from the Internet which gave us a list of Strings. Can we combine these? You betchya!

The lecture code provides a MovieDownloader class containing the exact same networking code utilized in the Appendix. We can then simply specify that the *model* String[] should be the result of the downloadMovieData() method, rather than manually created with a loop.

If you test this code, you'll notice that it doesn't work! The program will crash with a NetworkOnMainThreadException.

Android apps run by default on the ***Main Thread*** (also called the ***UI Thread***). This thread is in charge of all user interactions—handling button presses, scrolls, drags, etc.—but also UI *output* like drawing and displaying text! See Android Threads for more details.

- A thread is a piece of a program that is independently scheduled by the processor. Computers do exactly one thing at a time, but make it look like they are doing lots of tasks simultaneously by switching between them (i.e., between processes) really fast. Threads are a way that we can break up a single application or process into little "sub-process" that can be run simultaneously—by switching back and forth periodically so everyone has a chance to work

Within a single thread, all method calls are **synchronous**—that is, one has to finish before the next occurs. You can't get to step 4 without finishing step 3. With an event-driven system like Android, each method call is fast enough that this isn't a problem (you're done handling one click by the time the next occurs). But long, drawn-out processes like network access (or processing bitmaps, or accessing a database), could cause other tasks to have to wait. It's like a traffic jam!

- Tasks such as network access are **blocking** method calls, which stop the Thread from continuing. A blocked *Main Thread* will lead to the infamous **"Application not responding" (ANR)** error!

Thus we need to move the network code *off* the Main Thread, onto a **background thread**, thereby allowing it to run without blocking the user interaction that occurs on the Main

Thread. To do this, we will use a class called ASyncTask[15] to perform a task (such as network access) asynchronously—without waiting for other Threads.

Learning Android Development involves knowing about what classes exist, and can be used to solve problems, but how were we able to learn about the existing of this highly useful (and specialized) ASyncTask class? We started from the official API Guide on Processes and Threads Guide[16], which introduces this class! Thus to learn about new Android options, *read the docs*.

Note that an ASyncTask background thread will be *tied to the lifecycle of the Activity*: if we close the Activity, the network connection will die as well. A better but *much* more complex solution would be to use a Service—which is covered in a future lecture. But since this example just involves getting a small amount of data, we don't really care if the network connection gets dropped.

ASyncTask can be fairly complicated, but is a good candidate to practice learning from the API documentation. Looking at that documentation, the first thing you should notice (or would if the API was a little more readable) is that ASyncTask is **abstract**, meaning you'll need to *subclass* it in order to use it. Thus you can subclass it as an *inner* class inside the Activity that will use it (MovieDownloadTask is a good name).

You should also notice that ASyncTask is a *generic* class with three (3) generic parameters: the type of the Parameter to the task, the type of the Progress measurement reported by the task, and the type of the task's Result. We can fill in what types of Parameter and Result we want from our asynchronous method (e.g., take in a String and return a String[]), and use the Void type for the Progress measurement (since we won't be tracking that).

When we "run" an AsyncTask, it will do four (4) things, represented by four methods:

1. onPreExecute() is called *on the UI thread* before we run the task. This method can be used to perform any setup for the task.
2. doInBackground(Params...) is called *on the background thread* to do the work we want to be performed asynchronously. We **must** override this method (it's abstract!) The params and return type for the method need to match the ASyncTask generic types.
3. onProgressUpdate() can be indirectly called *on the UI thread* if we want to update our progress (e.g., update a progress bar). Note that UI changes can **only** be made on the UI thread!

4. onPostExecute(Result) is called *on the UI thread* to process any task results, which are passed as parameters to this method when doInBackground is finished.

The doInBackground() is what occurs on the background thread (and is the heart of the task), so we put our network accessing method call in there.

We can then *instantiate* a new ASyncTask object in the Activity's onCreate() callback, and call ASyncTask#execute(params) to start the task running on its own thread.

If you test this code, you'll notice that it still doesn't work! The program will crash with a SecurityException.

As a security feature, Android apps by default have very limited access to the overall operating system (e.g., to do anything other than show a layout). An app can't use the Internet (which might consume people's data plans!) without explicit permission from the user. This permission is given by the user at *install time*.

In order to get permission, the app needs to ask for it ("Mother may I…"). We do that by declaring that the app uses the Internet in the Manifest.xml file (which has all the details of our app!)

```
<uses-permission android:name="android.permission.INTERNET"/>
<!-- put this ABOVE the <application> tag -->
```

Note that Marshmallow introduced a new security model in which users grant permissions at *run-time*, not install time, and can revoke permissions whenever they want. To handle this, you need to add code to request "dangerous" permissions (like Location, Phone, or SMS access; Internet is *not* dangerous) each time you use it.

- For "normal" permissions (e.g., Internet), you declare the permission need in the Manifest.
- For "dangerous" permissions (e.g., Location), you declare the permission need in the Manifest **and** request permission programmatically in code each time you want to use it.

Once we've requested permission (and have been granted that permission by virtue of the user installing our application), we can finally connect to the Internet to download data. We can log out the request results to provide it.

In order to get the downloaded data into a ListView, we utilize the doPostExecute() method. This method is run on the *UI Thread* so we can use it to update the View (we can *only* change the View on the UI Thread, to avoid collisions). It also gets the results returned by doInBackground() passed to it!

We take that passed in String[] and put that into the ListView. Specifically, we feed it into the Adapter, which then works to populate the views.

- First clear out any previous data items in the adapter using adapter.clear().
- Then use adapter.add() or (adapter.addAll()) to add each of the new data items to the Adapter's model.
- You can call notifyDataSetChanged() on the Adapter to make sure that the View knows the data has changed, but this method is already called by the .add() method so isn't necessary in this situation.

To finalize the app: we can enable the user to search for different movies by copying the EditText and Button Views from the previous input_layout resource, accessing the text from the former when the later is pressed. We can then pass the EditText content String into the ASyncTask#execute() function (since we've declared that the generic ASyncTask takes that type as the first Parameter).

- We can actually pass in multiple String arguments using the String... params spread operator syntax (representing an arbitrary number of items of that type). See here for details. The value that the ASyncTask methods *actually* get is an array of the arguments.

In the end, we are able to downlod data from the Internet and show an interactive list of that data in the app! We've done a whirl-wind tour of Android in this process: Layouts in the XML, Adapters in the Activity, Threading in a new class, Security in the Manifest… bringing lots of parts together to provide a particular piece of functionality.

4.1 The Action Bar

Let's start one of the most prominent visual components in the default app: the ***App Bar*** or ***Action Bar***. This acts as the sort of "header" for your app, providing a dedicated space for navigation and interaction (e.g., through menus). The ActionBar[21] is a specific type of Toolbar that is most frequenly used as the App Bar, offering a particular "look and feel" common to Android applications.

While the AppCompatActivity used throughout this course automatically provides an Action Bar for the app, it is also possible to add it directly (such as if you are using a different Activity subclass). To add your own Action Bar, you specify a **theme** that does *not* include an ActionBar, and then include an <android.support.v7.window.Toolbar> element inside your layout wherever you want the toolbar to go. See Setting up the App Bar for details. This will also allow you to put the Toolbar anywhere in the application's layout (e.g., if you want it to be stuck to the bottom).

- To see this in action, change the android:theme attribute of the <application> element in the Manifest to "@style/Theme.AppCompat.Light.NoActionBar". We'll discuss this process in more detail when we talk about Themes and Styles.

From in the Activity's Java code, we can get access to the Action Bar by calling the getSupportActionBar() method (for a support Toolbar). We can then call utility methods on this object to interact with it; for example .hide() will hide the toolbar!

4.2 Menus

However, the main use for the Action Bar is a place to hold ***Menus***. A Menu (specifically, an **options menu**) is a set of items (think: buttons) that appear in the Action Bar. Menus can be specified both in the Activity and in a Fragment; if declared in both places, they are combined into a single menu in the Action Bar. This allows you to easily make "context-specific" options menus that are only available for an appropriate Fragment, while keeping Fragments modular and self-contained.

- *Fun fact:* before API 11, options menus appeared as buttons at the bottom of the screen!

Menus, like all other user-facing elements, are defined as XML resources, specifically of type **menu**. You can create a new menu resource through Android studio using File > New > Android resource file and then choosing the Menu Resource type. This will create an XML file with a main <menu> element.

Options can be added to the menu by specifying child XML elements, particularly <item> elements. Common <item> attributes include:

- **android:id**: a unique id used to refer to the specific option in the Java code

- **android:title** (**required** attribute): the text to display for the option. As user-facing text, the content should ideally be defined as an XML String resource.
- **app:showAsAction**: whether or not the option should be listed in the Action Bar, or collapsed under a "three-dots" button. Note when working with the appcompat library, this option uses the app namespace (instead of android); you will need to include this schema in the <menu> with the attribute xmlns:app="http://schemas.android.com/apk/res-auto".
- **android:icon**: an image to use when showing the option as a button on the menu **//CHECK THIS**

  You can use one of the many icons built into the Android, referenced as "@android:drawable/ic_*". Android Drawables[22] includes the full list, though not all drawables are publicly available through Android Studio.

- **android:orderInCategory**: used to order the item in the menu (or in a group). This acts as a "priority" (default 0; low comes first). Such prioritizing can be useful if you want to add suggestions about whether Fragment options should come before or after the Actiity options.

See the Menu resources guide[23] for the full list of options!

It is possible to include **one level** of sub-menus (a <menu> element inside an <item> element). Menu items can also be grouped together by placing them inside of a <group> element. All items in a group will be shown or hidden together, and can be further ordered within that group. Grouped icons can also be made checkable.

In order to show the menu in the running application, we need to tell the Action Bar which menu resource it should use (there may be a lot of resources). To do this, we override the onCreateOptionsMenu() callback in the Activity or Fragment, and then use the component's MenuInflater object to expand the menu:

```
public boolean onCreateOptionsMenu(Menu menu) {
   MenuInflater inflater = getMenuInflater();
   inflater.inflate(R.menu.main_menu, menu); //inflate into this menu
   return true;
}
```

- This procedure is similar in concept to how a Fragment's onViewCreated() method would inflate the Fragment into the Activity. In this case, the Menu is being inflated into the Action Bar.

We can respond to the menu items being selected by overriding the onOptionsItemSelected() callback. By convention, we use a switch on the item.getItemId() to determine what item was selected, and then act accordingly.

```
public boolean onOptionsItemSelected(MenuItem item) {
   switch(item.getItemId()){
      case R.id.menu_item1 :
         //do thing;
         return true;
      default:
         return super.onOptionsItemSelected(item);
   }
}
```

- On default (if the item selected isn't handled by any cases), we pass the callback up to super for "higher-level" components to check. For exampe, if a the menu option isn't handled by the Fragment (because the Fragment didn't add it), the event can be passed up through the Framework for eventually handling by the Activity (who did add it).
- This method should return true if the selection even has been handled (and thus should not be considered by anyone else). Return false if you want other components (e.g., other Fragments) to be able to respond to this option as well.

There are many other menu items that can be placed on Action Bar as well. We can also add Action Views that provide more complex interactions than just clicking buttons (for example, including a search bar). An Action Provider (like ShareActionProvider) is an action with its own customized layout, expanding into a separate View when clicked. We wil discuss how to utilize these features in a future lecture.

4.2.1 Context Menus

In addition to options menus available in the Action Bar, we can also specify contextual menus that pop up when the user long-presses on an element. This works similarly to using an options menu, but with a different set off callbacks:

- When setting up the the View layout (e.g., in an Activity's onCreate()), we specify that an element has a context menu using the registerForContextMenu() method, passing it the View we want to be able to create the menu for.
- Specify the context menu to use through the onCreateContextMenu() callback. This works exactly like setting up an options menu.
- In fact, a context menu can even use *the same menu* as an options menu! This reuse is one of the advantages of defining the user interface as XML.
- And mirroring the options menu, respond to context menu items being selected with the onContextItemSelected() callback.

This section has provided a very brief introduction to menus, but there are many more complex interactions that they support. I *highly* recommend that you read through the guide in order to learn what features may be available.

If you ever are using an app and wonder "how did they add this interface feature?", look it up! There is almost always a documented procedure and example for providing that kind of component.

4.3 Dialogs

While it is simple enough to make menu items that log out some text, logs cannot be seen the user. Instead, we woud like to show the message to the user as a kind of "pop-up" message.

A ***Dialog***[24] is a "pop-up" modal (a view which doesn't fill the screen) that either asks the user to make a decision or provides some additional information. At it's most basic, Dialogs are similar to the window.alert() function used in JavaScript.

There is a base Dialog class, but almost always we use a pre-defined subclass instead (similar to how we've use AppCompatActivity). AlertDialog[25] is the most common version: a simple message with buttons you can respond with (confirm, cancel, etc).

We don't actually instantiate an AlertDialog directly (in fact, it's constructors are protected so inaccessible to us). Instead we use a helper *factory* class called an AlertDialog.Builder. There are a number of steps to use a builder to create a Dialog:

1. Instantiate a new builder for this particular dialog. The constructor takes in a Context under which to create the Dialog. Note that once the builder is initialized, you can create and recreate the same dialog with a single method call—that's the benefits of using a factory.
2. Call "setter" methods on the builder in orer to specify the title, message, etc. for the dialog that will appear. This can be hard-coded text or a reference to an XML String resource (as a user-facing String, the later is more appropriate for published applications). Each setter method will return a reference to the builder, making it easy to chain them.

3. Use appropriate setter methods to specify callbacks (via a DialogInterface.OnClickListener) for individual buttons. Note that the "positive" button normally has the text "OK", but this can be customized.
4. Finally, actually instantiate the AlertDialog with the builder.create() method, using the **show()** method to make the dialog appear on the screen!

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Alert!")
      .setMessage("Danger Will Robinson!");
builder.setPositiveButton("I see it!", new DialogInterface.OnClickListener() {
  public void onClick(DialogInterface dialog, int id) {
    // User clicked OK button
  }
});

AlertDialog dialog = builder.create();
dialog.show();
```

An important part of learning to develop Android applications is being able to read the API to discover effective options. For example, can you read the AlertDialog.Builder API and determine how to add a "cancel" button to the alert?

While AlertDialog is the most common Dialog, Android supports other subclasses as well. For example, DatePickerDialog and TimePickerDialog provide pre-defined user interfaces for picking a date or a time respectively. See the Pickers guide for details about how to utilize these.

4.3.1 DialogFragments

The process described above will create and show a Dialog, but that dialog has a few problems in how it interacts with the rest of the Android framework—namely with the lifecycle of the Activity in which it is embedded.

For example, if the device changes configurations (e.g., is rotated from portrait to landscape) then the Activity is destroyed and re-created (it's onCreate() method will be called again). But if this happens while a Dialog is being shown, then a android.view.WindowLeaked error will be displayed and the Dialog is lost!

To avoid these problems, we need to have a way of giving that Dialog its own lifecycle which can interact with the the Activity's lifecycle… sort of like making it a *modular* piece of an Activity… that's right, we need to make it a Fragment! Specifically, we will use a subclass of Fragment called DialogFragment, which is a Fragment that displays as a modal dialog floating above the Activity (no extra work needed).

Just like with the Fragment examples from the previous lecture, we'll need to create our own subclass of DialogFragment. It's often easiest to make this a *nested class* if the Dialog won't be doing a lot of work (e.g., shows a simple confirmation).

Rathern than specifying a Fragment layout through onCreateView(), we can instead override the onCreateDialog() callback to specify a Dialog object that will provide the view hierarchy for the Fragment. This Dialog can be created with the AlertDialog.Builder class as before!

```
public static class MyDialogFragment extends DialogFragment {

    public static HelloDialogFragment newInstance() {
        Bundle args = new Bundle();
        HelloDialogFragment fragment = new HelloDialogFragment();
        fragment.setArguments(args);
        return fragment;
    }

    public Dialog onCreateDialog(Bundle savedInstanceState) {
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        //...
        AlertDialog dialog = builder.create();
```

```
        return dialog;
    }
}
```

Finally, we can actually show this DialogFragment by instantiating it (remember to use a newInstance() factory method!) and then calling the show() method on it to make it show as a Dialog. The show() method takes in a FragmentManager used to manage this transaction. By using a DialogFragment, it is possible to change the device configuration (rotate the phone) and the Dialog is retained.

Here's the other neat trick: a DialogFragment is just a Fragment. That means we can use it *anywhere* we normally used Fragments… including embedding them into layouts! For example if you made the MoviesFragment subclass DialogFragment instead of Fragment, it would be able to be used in the exact same as before. It's still a Fragment, just with extra features—one of which is a show() method that will show it as a Dialog!

- Use setStyle(DialogFragment.STYLE_NO_TITLE, android.R.style.Theme_Holo_Light_Dialog) to make the Fragment look a little more like a dialog.

The truth is that Dialogs are not very commonly used in Android (compare to other GU systems). Apps are more likely to just dynamically change the Fragment or Activity being shown, rather than interrupt the user flow by creating a pop-up modal. And 80% of the Dialogs that *are* used are AlertDialogs. Nevertheless, it is worth being familiar with this process and the patterns it draws upon!

4.4 Toasts

Dialogs are a powerful way of providing messages and information to users, but they are pretty "heavy" in terms of both their interaction (they stop all other interaction to show the user a message) and the effort required to implement them. Sometimes you just want a "pop-up" message that isn't quite as prominent and doesn't require the user to click "okay" once they've seen it.

A simple, quick way of giving some short visual feedback is to use what is called a **Toast**. This is a tiny little text box that pops up at the bottom of the screen for a moment to quickly display a message.

- It’s called a “Toast” because it pops up!

Toasts are pretty simple to implement, as with the following example (from the official documentation):

```
Context context = this; //getApplicationContext(); //use application context to avoid disappearing if Activity is closed quickly.
String text = "Hello toast!";
int duration = Toast.LENGTH_SHORT;

//use factory method instead of constructor
Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

But since this Activity *is* a Context, and we can just use the Toast anonymously, we can shorten this to a one-liner:

```
Toast.makeText(this, "Hello toast!", Toast.LENGTH_SHORT).show();
```

5.1 File Storage Locations

Android devices split file storage into two types: **Internal storage** and **External storage**. These names come from when devices had built-in memory as well as external SD cards, each of which may have had different interactions. However, with modern systems the “external storage” can refer to a section of a phone’s built-in memory as well; the distinctions are instead used for specifying *access* rather than physical data location.

- **Internal storage** is always accessible, and by default files saved internally are *only* accessible to your app. Similarly, when the user uninstalls your app, the internal files are deleted. This is usually the best place for “private” file data, or files that will only be used by your application.

- **External storage** is not always accessible (e.g., if the physical storage is removed), and is usually (but not always) *world-readable*. Normally files stored in External storage persist even if an app is uninstalled, unless certain options are used. This is usually used for “public” files that may be shared between applications.

When do we use each? Basically, you shuold use *Internal* storage for "private" files that you don't want to be available outside of the app, and use *External* storage otherwise.

- Note however that there are publicly-**hidden** *External* files—the big distinction between the storage locations is less visibility and more about *access*.

In addition, both of these storage systems also have a **"cache"** location (i.e., an *Internal Cache* and an *External Cache*). A cache is "(secret) storage for the future", but in computing tends to refer to "temporary storage". The Caches are different from other file storage, in that Android has the ability to automatically delete cached files if storage space is getting low… However, you can't rely on the operating system to do that on its own in an efficient way, so you should still delete your own Cache files when you're done with them! In short, use the Caches for temporary files, and try to keep them *small* (less than 1MB recommended).

- The user can easily clear an application's cache as well.

In code, using all of these storage locations involve working with the `File` class. This class represents a "file" (or a "directory") object, and is the same class you may be familiar with from Java SE.

- We can instantiate a `File` by passing it a directory (which is another `File`) and a filename (a `String`). Instantiating the file will create the file on disk (but empty, size 0) if it doesn't already exist.
- We can test if a `File` is a folder with the `.isDirectory()` method, and create new directories by taking a `File` and calling `.mkdir()` on it. We can get a list of `Files` inside the directory with the `listFiles()` method. See more API documentation for more details and options.

The difference between saving files to Internal and External storage, ***in practice***, simply involves which directory you put the file in! This lecture will focus on working with **External storage**, since that code ends up being a kind of "super-set" of implementation details needed for the file system in general. We will indicate what changes need to be made for interacting with Internal storage.

- This lecture will walk through implementing an application that will save whatever the user types into an text field to a file.

Because a device's External storage may be on removable media, in order to interact with it in any way we first need to check whether it is available (e.g., that the SD card is mounted). This can be done with the following check (written as a helper method so it can be reused):

```
public static boolean isExternalStorageWritable() {
  String state = Environment.getExternalStorageState();
  if (Environment.MEDIA_MOUNTED.equals(state)) {
    return true;
  }
  return false;
}
```

5.2 Permissions

Directly accessing the file system of any computer can be a significant security risk, so there are substantial protections in place to make sure that a malicious app doesn't run roughshod over a user's data. So in order to work with the file system, we first need to discuss how Android handles permissions in more detail.

One of the most import aspect of the Android operating system's design is the idea of **sandboxing**: each application gets its own "sandbox" to play in (where all its toys are kept), but isn't able to go outside the box and play with someone else's toys. The "toys" (components) parts that are outside of the sandbox are things that would be *impactful* to the user, such as network or file access. Apps are not 100% locked into their sandbox, but we need to do extra work to step outside.

- Sandboxing also occurs at a package level, where packages (applications) are isolated from packages *from other developers*; you can use certificate signing (which occurs as part of our build process automatically) to mark two packages as from the same developer if we want them to interact.
- Additionally, Android's underlying OS is Linux-based, so it actually uses Linux's permission system under the hood (with user and group ids that grant access to particular files or processes).

In order for an app to go outside of its sandbox (and use different components), it needs to request permission to leave. We ask for this permission ("Mother may I?") by declaring out-

of-sandbox usages explicitly in the Manifest, as we've done before with getting permission to access the Internet or send SMS messages.
Android permissions we can ask for are divided into two categories: normal and dangerous:

- **Normal permissions** are those that may impact the user (so require permission), but don't pose any serious risk. They are granted by the user at *install time*; if the user chooses to install the app, permission is granted to that app. See this list for examples of normal permissions. INTERNET is a normal permission.
- **Dangerous permissions**, on the other hand, have the risk of violating a user's privacy, or otherwise messing with the user's device or other apps. These permissions *also* need to be granted at install time. But ***IN ADDITION***, starting from Android 6.0 Marshmallow (API 23), users *additionally* need to grant dangerous permission access **at runtime**, when the app tries to actually invoke the "permitted" dangerous action.
  - The user grants permission via a system-generated pop-up dialog. Note that permissions are granted in "groups", so if the user agrees to give you RECEIVE_SMS permission, you get SEND_SMS permission as well. See the list of permission groups.
  - When the user grant permission at runtime, that permission stays granted as long as the app is installed. But the big caveat is that the user can choose to **revoke** or deny privileges at ***any*** time (they do this though System settings)! Thus you have to check *each time you want to access the feature* if the user has granted the privileges or not—you don't know if the user has *currently* given you permission, even if they had i

Writing to external storage is a *dangerous* permission, and thus we will need to do extra work to support the Marshmallow runtime permission system.

- In order to support runtime permissions, we need to specify our app's **target SDK** to be 23 or higher AND execute the app on a device running Android 6.0 (Marshmallow) or higher. Runtime permissions are only considered if the OS supports *and* the app is targeted that high. For lower-API devices or apps, permission is only granted at install time.

First we *still* need to request permission in the Manifest; if we haven't announced that we might ask for permission, we won't be allowed to ask in the future. In particular, saving files to External storage requires android.permission.WRITE_EXTERNAL_STORAGE permission (which will also grant us READ_EXTERNAL_STORAGE access).
Before we perform a dangerous action, we can check that we currently have permission:

```
int permissionCheck = ContextCompat.checkSelfPermission(activity, Manifest.permission.PERMISSION_NAME);
```

- This function basically "looks up" whether we've been granted a particular permission or not. It will return either PackageManager.PERMISSION_GRANTED or PackageManager.PERMISSION_DENIED.

If permission has been granted, great! We can go about our business (e.g., saving a file to external storage). But if permission has NOT been explicitly granted (at runtime), then we have to ask for it. We do this by calling:

```
ActivityCompat.requestPermissions(activity, new String[]{Manifest.permission.PERMISSION_NAME}, REQUEST_CODE);
```

- This method takes a context and then an *array* of permissions that we need access to (in case we need more than one). We also provide a request code (an int), which we can use to identify that particular request for permission in a callback that will be executed when the user chooses whether to give us access or not. This is the same pattern as when we sent an Intent for a *result*; asking for permission is conceptually like sending an Intent to the permission system!

We can then provide the callback that will be executed when the user decides whether to grant us permission or not:

```
public void onRequestPermissionsResult(int requestCode, String permissions[], int[] grantResults) {
 switch (requestCode) {
  case REQUEST_CODE:
   if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
```

```
      //have permission! Do stuff!
    }
  default:
      super.onRequestPermissionsResult(requestCode, permissions, grantResults);
  }
}
```

We check which request we're hearing the results for, what permissions were granted (if any—the user can piece-wise grant permissions), and then we can react if everything is good… like by finally saving our file!

- Note that if the user deny us permission once, we might want to try and explain *why* we're asking permission (see best practices) and ask again. Google offers a utility method (ActivityCompat#shouldShowRequestPermissionRationale()) which we can use to show a rationale dialog if they've denied us once. And if that's true, we might show a Dialog or something to explain ourselves–and if they OK that dialog, then we can ask again.

5.3 External Storage

Once we have permission to write to external file, we can actually do so! Since we've verified that the External storage is available, we now need to pick what directory in that storage to save the file in. With External storage, we have two options:

- We can save the file **publicly**. We use the getExternalStoragePublicDirectory() method to access a public directory, passing in what type of directory we want (e.g., DIRECTORY_MUSIC, DIRECTORY_PICTURES, DIRECTORY_DOWNLOADS etc). This basically drops files into the same folders that every other app is using, and is great for shared data and common formats like pictures, music, etc.. Files in the public directories can be easily accessed by other apps (assuming the app has permission to read/write from External storage!)
- Alternatively starting from API 18, we save the file **privately**, but still on External storage (these files *are* world-readable, but are hidden from the user as media, so they don't "look" like public files). We access this directory with

the getExternalFilesDir() method, again passing it a *type* (since we're basically making our own version of the public folders). We can also use null for the type, giving us the root directory.

Since API 19 (4.4 KitKat), you don't need permission to write to *private* External storage. So you can specify that you only need permission for versions lower than that:

```xml
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"
android:maxSdkVersion="18" />
```

We can actually look at the emulator's file-system and see our files by created using adb. Connect to the emulator from the terminal using adb -s emulator-5554 shell (note: adb needs to be on your PATH). **Public** external files can usually be found in /storage/sdcard/Folder, while **private** external files can be found in /storage/sdcard/Android/data/package.name/files (these paths may vary on different devices).

Once we've opened up the file, we can write content to it by using the same IO classes we've used in Java:

- The "low-level" way to do this is to create a a FileOutputStream object (or a FileInputStream for reading). We just pass this constructor the File to write to. We write bytes to this stream… but can write a String by calling myString.getBytes(). For reading, we'll need to read in *all* the lines/characters, and probably build a String out of them to show. This is actually the same loop we used when reading data from an HTTP request!
- However, we can also use the same *decorators* as in Java (e.g., BufferedReader, PrintWriter, etc.) if we want those capabilities; it makes reading and writing to file a little easier
- In either case, **remember to .close() the stream when done** (to avoid memory leaks)!

```
//writing
try {
   //saving in public Documents directory
   File dir = getExternalStoragePublicDirectory(Environment.DIRECTORY_DOCUMENTS);
```

```
    if (!dir.exists()) { dir.mkdirs(); } //make dir if doesn't otherwise exist
    File file = new File(dir, FILE_NAME);
    Log.v(TAG, "Saving to  " + file.getAbsolutePath());

    PrintWriter out = new PrintWriter(new FileWriter(file, true));
    out.println(textEntry.getText().toString());
    out.close();
} catch (IOException ioe) {
    Log.d(TAG, Log.getStackTraceString(ioe));
}

//reading
try {
    File dir = getExternalStoragePublicDirectory(Environment.DIRECTORY_DOCUMENTS);
    File file = new File(dir, FILE_NAME);
    BufferedReader reader = new BufferedReader(new FileReader(file));
    StringBuilder text = new StringBuilder();

    //read the file
    String line = reader.readLine();
    while (line != null) {
        text.append(line + "\n");
        line = reader.readLine();
    }

    textDisplay.setText(text.toString());
    reader.close();
} catch (IOException ioe) {
    Log.d(TAG, Log.getStackTraceString(ioe));
}
```

This will allow us to have our “save” button write the message to the file, and have our “read” button load the message from the file (and display it on the screen)!

5.4 Internal Storage & Cache

Internal storage works pretty much the same way as External storage. Remember that Internal storage is always *private* to the app. We also don't need permission to access Internal storage!

For Internal storage, we can use the getFilesDir() method to access to the files directory (just like we did with External storage). This method normally returns the folder at /data/data/package.name/files.
Alternatively, we can use Context#openFileOutput() (or Context#openFileInput()) and pass it the *name* of the file to open. This gives us back the Stream object for that file in the Internal storage file directory, without us needing to do any extra work (cutting out the middle-man!)

- These methods take a second parameter: MODE_PRIVATE will create the file (or *replace* a file of the same name). Other modes available are: MODE_APPEND (which adds to the end of the file if it exists instead of erasing). ~~MODE_WORLD_READABLE~~, and ~~MODE_WORLD_WRITEABLE~~ are deprecated.
- Note that you can wrap a FileInputStream in a InputStreamReader in a BufferedReader.

We can access the Internal Cache directory with getCacheDir() (and same read/write process), or the External Cache directory with getExternalCacheDir(). We almost always use the Internal Cache, because why would you want temporary files to be world-readable (other than maybe temporary images…)
And again, once you have the file, you use the same process for reading and writing as External storage.

For practice make the provided toggle support reading and writing to an Internal file as well. This will of course be *different* file than that used with the External switch. Ideally this code could be refactored to avoid duplication, but it gets tricky with the need for checked exception handling.

5.5 Example: Saving Pictures

As another example of how we might use the storage system, consider the “take a selfie” system from lecture 8. The code for taking a piecture can be found in a separate PhotoActivity (which is accessible via the options menu).

To review: we sent an Intent with the MediaStore.ACTION_IMAGE_CAPTURE action, and the *result* of that Intent included an *Extra* that was a BitMap of a low-quality thumbnail for the image. But if we want to save a higher resolution version of that picture, we’ll need to store that image in the file system!

To do this, we’re actually going to modify the Intent we *send* so it includes an additional Extra: a file in which the picture data can be saved. Effectively, we’ll have *our Activity* allocate some memory for the picture, and then tell the Camera where it can put the picture data that it captures. (Intent envelops are too small to carry entire photos around!)

Before we send the Intent, we’re going to go ahead and create an (empty) file:

```
File file = null;
try {
   String timestamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date()); //include timestamp

   //ideally should check for permission here, skipping for time
   File dir = Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES);
   file = new File(dir, "PIC_"+timestamp+".jpg");
   boolean created = file.createNewFile(); //actually make the file!
   Log.v(TAG, "File created: "+created);

} catch (IOException ioe) {
   Log.d(TAG, Log.getStackTraceString(ioe));
}
```

We will then specify an additional Extra to give that file’s location to the camera: if we use MediaStore.EXTRA_OUTPUT as our Extra’s *key*, the camera will know what to do with that! However, the extra won’t actually be the File but a **Uri** (recall: the “url” or location of a file). We’re not sending the file itself, but the ***location*** of that file (because it’s smaller data to fit in the Intent envelope).

- We can get this Uri with the Uri.fromFile(File) method:

- //*save as instance variable to access later when picture comes back*
  pictureFileUri = Uri.fromFile(file);
- Then when we get the picture result back from the Camera (in our onActivityResult callback), we can access that file at the saved Uri and use it to display the image! The ImageView.setImageUri() is a fast way of showing an image file.

Note that when working with images, we can very quickly run out of memory (because images can be huge). So we'll often want to "scale down" the images as we load them into memory. Additionally, image processing can take a while so we'd like to do it off the main thread (e.g., in an AsyncTask). This can become complicated; the recommended solution is to use a third-party library such as Glide, Picasso, or Fresco.

5.6 Sharing Files

Once we have a file storing the image, we can also save that image with other apps!

As always, in order to interact with other apps, we use an Intent. We can craft an *implicit intent* for ACTION_SEND, sending a message to any apps that are able to send (share) pictures. We'll set the data type as image/* to mark this as an image. We will also attach the file as an extra (specifically an EXTRA_STREAM). Again note that we don't actually put the *file* in the extra, but rather tha **Uri** for the file!
Since multiple activities may support this action, we can wrap the intent in a "chooser" to force the user to pick which Activity to use:

```
Intent chooser = Intent.createChooser(shareIntent, "Share Picture");
//check that there is at least one option
if (shareIntent.resolveActivity(getPackageManager()) != null) {
  startActivity(chooser);
}
```

There is one complication though: because we're saving files in External storage, the app who is executing the ACTION_SEND will need to have permission to read the file (e.g., to access External storage). The Messenger app on the emulator appears to lack this permission by default, though we need to take a slightly different approach:
Rather than putting the file:// Uri in the Intent's extra, we'll need to create a content:// Uri for a *ContentProvider* who is able to provide files to anyone who requests them regardless

of permissions (the provider grants permission to access its content). Luckily, each image stored in the public directories is automatically tracked by a ContentProvider known as the MediaStore. It easy to fetch a content:// Uri for a particular image file from this provider:

```
MediaScannerConnection.scanFile(this, new String[] {file.toString()}, null,
        new MediaScannerConnection.OnScanCompletedListener() {
            public void onScanCompleted(String path, Uri uri) {
                mediaStoreUri = uri;  //save the content:// Uri for later
                Log.v(TAG, "MediaStore Uri: "+uri);
            }
        });
```

This provides a Uri that can be given to the Intent, and that the Messenger app will be able to access! We can generate this Uri as soon as we have a file for the image to be saved in.

5.6.1 Bonus: Sharing with a FileProvider

This section has not be edited for formatting or content.

What happens if we try and share an Internal file? You'll get an error (actually notified the use!), because the other (email) app doesn't have permission to read that file!

- There is a way around this though, and it's by using a ContentProvider (haha!) A ContentProvider explicit is about making content available outside of a package (that's why we declared it in the Manifest). Specifically, a ContentProvider can convert a set of Files into a set of data contents (e.g., accessible with the content:// protocol) that can be used and returned and understood by other apps!
    - Kind of like a "File Server"
- Android includes a FileProvider class in the support library that does exactly this work.

Setting up a FileProvider is luckily not too complex, though it has a couple of steps. You will need to declare the <provider> inside you Manifest (see the guide link for an example).

```
<provider
   android:name="android.support.v4.content.FileProvider"
   android:authorities="edu.uw.mapdemo.fileprovider"
   android:exported="false"
   android:grantUriPermissions="true">
```

```
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/fileprovider" />
</provider>
```

The attributes you will need to specify are:

- android:authority should be your package name followed by .fileprovider (e.g., edu.uw.myapp.fileprovider). This says what source/domain is granting permission for others to use the file.
- The child <meta-data> tag includes an androd:resource attribute that should point to an XML resource, of type xml (the same as used for your SharedPreferences). *You will need to create this file!* The contents of this file will be a list of what *subdirectories* you want the FileProvider to be able to provide. It will look something like:

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <files-path name="my_maps" path="maps/" />
</paths>
```

The <files-path> entry refers to a subdirectory inside the Internal Storage files (the same place that .getFilesDir() points to), with the path specifying the name of the subdirectory (see why we made one called maps/?)

Once you have the provider specified, you can use it to get a Uri to the “shared” version of the file using:

```
Uri fileUri = FileProvider.getUriForFile(context, "edu.uw.myapp.fileprovider", fileToShare);
```